RAPIDS

End-to-end data-science and geospatial analytics with GPUs, RAPIDS, and Apache Arrow

Joshua Patterson - GM, Data Science
RAPIDS
End-to-End Accelerated GPU Data Science

Data Preparation -> Model Training -> Visualization

Dask

cuDF, cuIO, cuML, cuGraph, PyTorch, Chainer, MxNet, cuXfilter, pyViz

GPU Memory

Apache Arrow
Data Processing Evolution
Faster data access, less data movement

Hadoop Processing, Reading from disk
- HDFS Read
- Query
- HDFS Write
- HDFS Read
- ETL
- HDFS Write
- HDFS Read
- ML Train

Spark In-Memory Processing
- HDFS Read
- Query
- ETL
- ML Train

Traditional GPU Processing
- HDFS Read
- GPU Read
- Query
- CPU Read
- GPU Read
- ETL
- GPU Read
- CPU Read
- GPU Read
- ML Train

- 25-100x Improvement
- Less code
- Language flexible
- Primarily In-Memory

- 5-10x Improvement
- More code
- Language rigid
- Substantially on GPU
Data Movement and Transformation

The bane of productivity and performance
Data Movement and Transformation

What if we could keep data on the GPU?
Learning from Apache Arrow

- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (e.g., Parquet-to-Arrow reader)

*From Apache Arrow Home Page - https://arrow.apache.org/*
Data Processing Evolution
Faster data access, less data movement

Hadoop Processing, Reading from disk
- HDFS Read
- Query
- HDFS Write
- HDFS Read
- ETL
- HDFS Write
- HDFS Read
- ML Train

Spark In-Memory Processing
- HDFS Read
- Query
- ETL
- ML Train

Traditional GPU Processing
- HDFS Read
- GPU Read
- Query
- GPU Read
- CPU Write
- ETL
- CPU Write
- GPU Read
- ML Train

RAPIDS
- Arrow Read
- Query
- ETL
- ML Train

5-10x Improvement
More code
Language rigid
Substantially on GPU

50-100x Improvement
Same code
Language flexible
Primarily on GPU

25-100x Improvement
Less code
Language flexible
Primarily In-Memory
RAPIDS Core
Open Source Data Science Ecosystem

Familiar Python APIs

Data Preparation → Model Training → Visualization

Dask

Pandas Analytics
Scikit-Learn Machine Learning
NetworkX Graph Analytics
PyTorch Chainer MxNet Deep Learning
Matplotlib/Seaborn Visualization

CPU Memory
RAPIDS
End-to-End Accelerated GPU Data Science

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics → cuML Machine Learning → cuGraph Graph Analytics → PyTorch Chainer MxNet Deep Learning → cuXfilter <-> pyViz Visualization

GPU Memory

Apache Arrow
Dask
RAPIDS
Scaling RAPIDS with Dask
Why Dask?

**PyData Native**
- **Easy Migration**: Built on top of NumPy, Pandas, Scikit-Learn, etc.
- **Easy Training**: With the same APIs
- **Trusted**: With the same developer community

**Deployable**
- **HPC**: SLURM, PBS, LSF, SGE
- **Cloud**: Kubernetes
- **Hadoop/Spark**: Yarn

**Easy Scalability**
- Easy to install and use on a laptop
- Scales out to thousand-node clusters

**Popular**
- Most common parallelism framework today in the PyData and SciPy community
Why OpenUCX?
Bringing hardware accelerated communications to Dask

• TCP sockets are slow!

• UCX provides uniform access to transports (TCP, InfiniBand, shared memory, NVLink)

• Python bindings for UCX (ucx-py) in the works

• Will provide best communication performance, to Dask based on available hardware on nodes/cluster
Scale up with RAPIDS

**RAPIDS and Others**
Accelerated on single GPU
- NumPy -> CuPy/PyTorch/..
- Pandas -> cuDF
- Scikit-Learn -> cuML
- Numba -> Numba

**PyData**
- NumPy, Pandas, Scikit-Learn, Numba and many more
- Single CPU core
- In-memory data
Scale out with RAPIDS + Dask with OpenUCX

**RAPIDS and Others**
Accelerated on single GPU
- NumPy -> CuPy/PyTorch/...
- Pandas -> cuDF
- Scikit-Learn -> cuML
- Numba -> Numba

**RAPIDS + Dask with OpenUCX**
Multi-GPU
- On single Node (DGX)
- Or across a cluster

**PyData**
NumPy, Pandas, Scikit-Learn, Numba and many more
- Single CPU core
- In-memory data

**Dask**
Multi-core and Distributed PyData
- NumPy -> Dask Array
- Pandas -> Dask DataFrame
- Scikit-Learn -> Dask-ML
- ... -> Dask Futures
cuDF
RAPIDS
GPU Accelerated data wrangling and feature engineering

Data Preparation → Model Training → Visualization

Dask

cuDF, cuIO, Analytics

cuML, Machine Learning

cuGraph, Graph Analytics

PyTorch, Chainer, MxNet, Deep Learning

cuXfilter <-> pyViz, Visualization

GPU Memory → Apache Arrow
ETL - the Backbone of Data Science

libcuDF is...

CUDA C++ Library

- Low level library containing function implementations and C/C++ API

- Importing/exporting Apache Arrow in GPU memory using CUDA IPC

- CUDA kernels to perform element-wise math operations on GPU DataFrame columns

- CUDA sort, join, groupby, reduction, etc. operations on GPU DataFrames

```c++
void some_function( cudf::column const* input,
                    cudf::column * output,
                    args...)
{
    // Do something with input
    // Produce output
}
```
ETL - the Backbone of Data Science

cuDF is...

Python Library

- A Python library for manipulating GPU DataFrames following the Pandas API
- Python interface to CUDA C++ library with additional functionality
- Creating GPU DataFrames from Numpy arrays, Pandas DataFrames, and PyArrow Tables
- JIT compilation of User-Defined Functions (UDFs) using Numba
Benchmarks: single-GPU Speedup vs. Pandas

**Benchmarks:** single-GPU Speedup vs. Pandas

**cuDF v0.9, Pandas 0.24.2**

Running on NVIDIA DGX-1:

**GPU:** NVIDIA Tesla V100 32GB  
**CPU:** Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

**Benchmark Setup:**

DataFrames: 2x int32 columns key columns,  
3x int32 value columns

Merge: inner

GroupBy: count, sum, min, max calculated for each value column
ETL - the Backbone of Data Science

String Support

**Current v0.9 String Support**
- Regular Expressions
- Element-wise operations
  - Split, Find, Extract, Cat, Typecasting, etc...
- String GroupBys, Joins
- Categorical columns fully on GPU

**Future v0.10+ String Support**
- Combining cuStrings into libcudf
- Extensive performance optimization
- More Pandas String API compatibility
- JIT-compiled String UDFs
Extraction is the Cornerstone

**culIO for Faster Data Loading**

- Follow Pandas APIs and provide >10x speedup
- CSV Reader - v0.2, CSV Writer v0.8
- Parquet Reader - v0.7, Parquet Writer v0.10
- ORC Reader - v0.7, ORC Writer v0.10
- JSON Reader - v0.8
- Avro Reader - v0.9

- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!

- Key is GPU-accelerating both parsing and decompression wherever possible

```python
1]: import pandas, cudf
2]: %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
    CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
    Wall time: 29.2 s
2]: 12748986
3]: %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
    CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
    Wall time: 2.12 s
3]: 12748986
4]: !du -hs data/nyc/yellow_tripdata_2015-01.csv
1.9G  data/nyc/yellow_tripdata_2015-01.csv
```

Source: Apache Crail blog: SQL Performance: Part 1 - Input File Formats
ETL is not just DataFrames!
RAPIDS
Building bridges into the array ecosystem

Data Preparation -> Model Training -> Visualization

Dask

cuDF cuIO Analytics -> cuML Machine Learning -> cuGraph Graph Analytics

PyTorch Chainer MxNet Deep Learning

cuXfilter <> pyViz Visualization

GPU Memory

Apache Arrow
Interoperability for the Win
DLPack and __cuda_array_interface__

PyTorch
mpi4py
mxnet
Numba
Chainer
CuPy
Interoperability for the Win

DLPack and __cuda_array_interface__

PYTORCH

mpi4py

mxnet

RAPIDS

Open GPU Data Science

Numba

Chainer

CuPy
ETL - Arrays and DataFrames

Dask and CUDA Python arrays

- Scales NumPy to distributed clusters
- Used in climate science, imaging, HPC analysis up to 100TB size
- Now seamlessly accelerated with GPUs
Benchmark: single-GPU CuPy vs NumPy

### Also...Achievement Unlocked:

**Petabyte Scale Data Analytics with Dask and CuPy**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single CPU Core</td>
<td>2hr 39min</td>
</tr>
<tr>
<td>Forty CPU Cores</td>
<td>11min 30s</td>
</tr>
<tr>
<td>One GPU</td>
<td>1min 37s</td>
</tr>
<tr>
<td>Eight GPUs</td>
<td>19s</td>
</tr>
</tbody>
</table>

### 3.2 petabytes in less than 1 hour

*Cluster configuration: 20x GCP instances, each instance has:*

**CPU:** 1 VM socket (Intel Xeon CPU @ 2.30GHz), 2-core, 2 threads/core, 132GB mem, GbE ethernet, 950 GB disk

**GPU:** 4x NVIDIA Tesla P100-16GB-PCIe (total GPU DRAM across nodes 1.22 TB)

**Software:** Ubuntu 18.04, RAPIDS 0.5.1, Dask=1.1.1, Dask-Distributed=1.1.1, CuPY=5.2.0, CUDA 10.0.130

---

https://blog.dask.org/2019/01/03/dask-array-gpus-first-steps
Machine Learning
More models more problems
Problem
Data sizes continue to grow

Massive Dataset

- Histograms / Distributions
- Dimension Reduction
  - Feature Selection
- Remove Outliers
- Sampling

Better to start with as much data as possible and explore / preprocess to scale to performance needs.

Time Increases

Hours? Days?

Iterate. Cross Validate & Grid Search. Iterate some more.

Meet reasonable speed vs accuracy tradeoff
ML Technology Stack

- Python
- Cython
- cuML Algorithms
- cuML Prims
- CUDA Libraries
- CUDA
- Dask cuML
- Dask cuDF
- cuDF
- Numpy
- Thrust
- Cub
- cuSolver
- nvGraph
- CUTLASS
- cuSparse
- cuRand
- cuBlas
Algorithms
GPU-accelerated Scikit-Learn

- Classification / Regression
  - Decision Trees / Random Forests
  - Linear Regression
  - Logistic Regression
  - K-Nearest Neighbors
- Inference
  - Random forest / GBDT inference
- Clustering
  - K-Means
  - DBSCAN
  - Spectral Clustering
- Decomposition & Dimensionality Reduction
  - Principal Components
  - Singular Value Decomposition
  - UMAP
  - Spectral Embedding
- Time Series
  - Holt-Winters
  - Kalman Filtering

Key:
- Preexisting
- NEW for 0.9

Cross Validation
Hyper-parameter Tuning
More to come!
RAPIDS matches common Python APIs

CPU-Based Clustering

```python
from sklearn.datasets import make_moons
import pandas

X, y = make_moons(n_samples=int(1e2),
noise=0.05, random_state=0)

X = pandas.DataFrame({'fe%d%i: X[:, i]
for i in range(X.shape[1])})

from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
```
RAPIDS matches common Python APIs

**GPU-Accelerated Clustering**

```python
from sklearn.datasets import make_moons
import cudf

X, y = make_moons(n_samples=int(1e2), noise=0.05, random_state=0)

X = cudf.DataFrame({'fea%d' % i: X[:, i] for i in range(X.shape[1])})

def make_moons(n_samples=100, noise=0.05, random_state=0):
    """Generate `n_samples` samples with two Gaussian centers."""
    if random_state is not None:
        random_state = np.random.RandomState(random_state)
    X = np.zeros((n_samples, 2))
    X[:, 0] = np.random.randn(n_samples)
    X[:, 1] = 0.5 * np.tan(2 * np.pi * X[:, 0]) + np.random.normal(scale=noise, size=n_samples)
    X += np.random.randn(n_samples, 2) * noise
    return X, np.zeros(n_samples)

from cuml import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
```

Find Clusters

```python
from sklearn.datasets import make_moons
import cudf

X, y = make_moons(n_samples=int(1e2), noise=0.05, random_state=0)

X = cudf.DataFrame({'fea%d' % i: X[:, i] for i in range(X.shape[1])})

def make_moons(n_samples=100, noise=0.05, random_state=0):
    """Generate `n_samples` samples with two Gaussian centers."""
    if random_state is not None:
        random_state = np.random.RandomState(random_state)
    X = np.zeros((n_samples, 2))
    X[:, 0] = np.random.randn(n_samples)
    X[:, 1] = 0.5 * np.tan(2 * np.pi * X[:, 0]) + np.random.normal(scale=noise, size=n_samples)
    X += np.random.randn(n_samples, 2) * noise
    return X, np.zeros(n_samples)

from cuml import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
```
Benchmarks: single-GPU cuML vs scikit-learn

1x V100 vs
2x 20 core CPU
Road to 1.0
August 2019 - RAPIDS 0.9

<table>
<thead>
<tr>
<th>cuML</th>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Boosted Decision Trees (GBDT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logistic Regression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Forest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Means</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBSCAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Holt-Winters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kalman Filter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t-SNE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Principal Components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singular Value Decomposition</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Road to 1.0

**March 2020 - RAPIDS 0.14**

<table>
<thead>
<tr>
<th>cuML</th>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Boosted Decision Trees (GBDT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logistic Regression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Forest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Means</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBSCAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIMA &amp; Holt-Winters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kalman Filter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t-SNE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Principal Components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singular Value Decomposition</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cuGraph
Graph Analytics
More connections more insights
GOALS AND BENEFITS OF CUGRAPH
Focus on Features and User Experience

Breakthrough Performance
- Up to 500 million edges on a single 32GB GPU
- Multi-GPU support for scaling into the billions of edges

Multiple APIs
- **Python**: Familiar NetworkX-like API
- **C/C++**: lower-level granular control for application developers

Seamless Integration with cuDF and cuML
- Property Graph support via DataFrames

Growing Functionality
- Extensive collection of algorithm, primitive, and utility functions
Graph Technology Stack

* nvGRAPH has been Opened Sourced and integrated into cuGraph. A legacy version is available in a RAPIDS GitHub repo

* Gunrock is from UC Davis
Algorithms

GPU-accelerated NetworkX

- Community
  - Spectral Clustering
  - Balanced-Cut
  - Modularity Maximization
  - Louvain
- Components
  - Subgraph Extraction
  - Triangle Counting
- Link Analysis
- Link Prediction
  - Jaccard
  - Weighted Jaccard
  - Overlap Coefficient
- Traversal
  - Single Source Shortest Path (SSSP)
  - Breadth First Search (BFS)
- Structure
  - COO-to-CSR (Multi-GPU)
  - Transpose
  - Renumbering

More to come!

Utilities
  - Query Language

Multi-GPU
G = cugraph.Graph()
G.add_edge_list(gdf[“src_0”], gdf[“dst_0”], gdf[“data”])
df, mod = cugraph.nvLouvain(G)

Louvain returns:
cudf.DataFrame with two names columns:
louvain[“vertex”]: The vertex id.
louvain[“partition”]: The assigned partition.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>preferentialAttachment</td>
<td>100,000</td>
<td>999,970</td>
</tr>
<tr>
<td>caidaRouterLevel</td>
<td>192,244</td>
<td>1,218,132</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>299,067</td>
<td>299,067</td>
</tr>
<tr>
<td>dblp-2010</td>
<td>326,186</td>
<td>1,615,400</td>
</tr>
<tr>
<td>citationCiteseer</td>
<td>268,495</td>
<td>2,313,294</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>540,486</td>
<td>30,491,458</td>
</tr>
<tr>
<td>coPapersCiteseer</td>
<td>434,102</td>
<td>32,073,440</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>1,696,415</td>
<td>22,190,596</td>
</tr>
</tbody>
</table>
## Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

<table>
<thead>
<tr>
<th>HiBench Scale</th>
<th>Vertices</th>
<th>Edges</th>
<th>CSV File (GB)</th>
<th># of GPUs</th>
<th>PageRank for 3 Iterations (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huge</td>
<td>5,000,000</td>
<td>198,000,000</td>
<td>3</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>BigData</td>
<td>50,000,000</td>
<td>1,980,000,000</td>
<td>34</td>
<td>3</td>
<td>5.1</td>
</tr>
<tr>
<td>BigData x2</td>
<td>100,000,000</td>
<td>4,000,000,000</td>
<td>69</td>
<td>6</td>
<td>9.0</td>
</tr>
<tr>
<td>BigData x4</td>
<td>200,000,000</td>
<td>8,000,000,000</td>
<td>146</td>
<td>12</td>
<td>18.2</td>
</tr>
<tr>
<td>BigData x8</td>
<td>400,000,000</td>
<td>16,000,000,000</td>
<td>300</td>
<td>16</td>
<td>31.8</td>
</tr>
</tbody>
</table>
## Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

<table>
<thead>
<tr>
<th>HiBench Scale</th>
<th>Vertices</th>
<th>Edges</th>
<th>CSV File (GB)</th>
<th># of GPUs</th>
<th>PageRank for 3 Iterations (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huge</td>
<td>5,000,000</td>
<td>198,000,000</td>
<td>3</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>BigData</td>
<td>50,000,000</td>
<td>1,980,000,000</td>
<td>34</td>
<td>3</td>
<td>5.1</td>
</tr>
<tr>
<td>BigData x2</td>
<td>100,000,000</td>
<td>4,000,000,000</td>
<td>69</td>
<td>6</td>
<td>9.0</td>
</tr>
<tr>
<td>BigData x4</td>
<td>200,000,000</td>
<td>8,000,000,000</td>
<td>146</td>
<td>12</td>
<td>18.2</td>
</tr>
<tr>
<td>BigData x8</td>
<td>400,000,000</td>
<td>16,000,000,000</td>
<td>300</td>
<td>16</td>
<td>31.8</td>
</tr>
</tbody>
</table>

BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX ⇒ 96 mins!
Road to 1.0
August 2019 - RAPIDS 0.9

<table>
<thead>
<tr>
<th>cuGraph</th>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard and Weighted Jaccard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subgraph Extraction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katz Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak and Strong)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louvain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InfoMap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Cores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cuGraph</td>
<td>Single-GPU</td>
<td>Multi-GPU</td>
<td>Multi-Node-Multi-GPU</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------------</td>
<td>-----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Jaccard and Weighted Jaccard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subgraph Extraction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katz Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak and Strong)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louvain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InfoMap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Cores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RAPIDS Geospatial Applications
RAPIDS Geospatial Applications

cuGraph SSSP

Read the source data and adjust the vertex IDs

```python
# Import needed libraries
import cudf
import numpy as np

# Test file - Read OS Open Roads as a graph. (Store dataset in \data\folder)
datafile = '/data/road_graph/gb_read_graph_20190528.csv'

data = pd.read_csv(datafile)

cols = ['src', 'dst', 'length', 'drivetime', 'type', 'x1', 'y1', 'x2', 'y2']
dtypes = ['int32', 'int32', 'float32', 'float32', 'float32', 'int32', 'float64', 'float64', 'float64']
gdf = cudf.read_csv(datafile, names=cols, dtype=dtypes, sep=';')

c = gdf['src']
d = gdf['dist']

c = c - d

data['src'] = c

data['dist'] = d

data['src'].unique()
data['dist'].unique()

# Display the results
print(data.head())
```

Example Isochrone Zones on Mainland GB Road Network

Produced using cuGraph in RAPIDS
Load the stores list and calculate one hour drive times

```r
# Load stores list
data_file <- "data/road_graph/uk_stores.csv"
cols <- c("ID", "y")
dtypes <- c("int32", "float64", "float64")
sdf <- cuDF_read_csv(data_file, names = cols, dtypes = dtypes, skiprows = 1)
stores = sdf["ID", to_array()], sdf["y", to_array()], sdf["y", to_array()]

# Print stores list
print(sdf)
```

<table>
<thead>
<tr>
<th>ID</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25257</td>
<td>324049.49</td>
</tr>
<tr>
<td>1</td>
<td>22739</td>
<td>1440087.59</td>
</tr>
<tr>
<td>2</td>
<td>93327</td>
<td>368448.00</td>
</tr>
<tr>
<td>3</td>
<td>100362</td>
<td>277367.0</td>
</tr>
<tr>
<td>4</td>
<td>103272</td>
<td>216041.00</td>
</tr>
<tr>
<td>5</td>
<td>123542</td>
<td>339183.12</td>
</tr>
<tr>
<td>6</td>
<td>145556</td>
<td>311452.83</td>
</tr>
<tr>
<td>7</td>
<td>146559</td>
<td>375000.99</td>
</tr>
<tr>
<td>8</td>
<td>275050</td>
<td>215558.02</td>
</tr>
<tr>
<td>9</td>
<td>457087</td>
<td>448150.53</td>
</tr>
</tbody>
</table>

[08 more rows]

```r
# Time to drive
store_drivetimes <- []
# for each store find the drive times to all vertices within 1 hour drive time by filtering (initially on 70 miles)
for i in range(len(stores)):
    # initial filter on 70 miles (112054 metres) grid from store node
    query_x = "((x>="store[stores[1][1]]-112054") and x="store[stores[1][1]]+112054")" or "(x="store[stores[1][1]]-112054") and y="store[stores[1][1]]+112054")" and "(y="store[stores[1][1]]+112054") and y="store[stores[1][1]]+112054")"
    query_y = "((y="store[stores[1][1]]-112054") and y="store[stores[1][1]]+112054")" or "(y="store[stores[1][1]]-112054") and y="store[stores[1][1]]+112054")" and "(y="store[stores[1][1]]+112054") and y="store[stores[1][1]]+112054")"
    sdf = gdf.query("(query_x)" and "(query_y)")
    gdf = gdf.query("(x)" and "(y)")
    # create a road Graph weighted on drive time
    G = cuGraph.Graph()
    G.add_edge_list(sdf["src", 0], sdf["dst", 0], sdf["drivetime"])
    # Call cuGraph.sssp to get the drivetime from store vertex
df = cuGraph.sssp(G, stores[1][1])
    # Filter 1 hour (3600 seconds) and save results
    store_drivetimes += [df.query("(drivetime>3600)")]
```

CPU time: user 45.7 s, sys: 10.7 s, total: 56.4 s
Wall time: 56.4 s
RAPIDS Geospatial Applications

cuML K-means

```python
# Pearson Correlation Coefficient
class stats_df(cudf.DataFrame):
    def pearson_corr(self, c1, c2):
        c1_n = self[c1].mean()
        c2_n = self[c2].mean()
        c1_s = self[c1].std()
        c2_s = self[c2].std()
        n = self[c1].count()
        return ((self[c1] * self[c2]).sum() - n*c1_n*c2_n) / ((n-1) * c1_s * c2_s)

    def prediction_interval(self, c, c1, c2):
        n = self[c].count()
        s = self[c].std()
        return (m - t1.06, m + t1.06)

df.__class__ = stats_df

df.pearson_corr('P_INFANT', 'P_75+')
print(np.array(df.prediction_interval('P_75+'))

m = (df['P_INFANT'] * df['P_75+']).mean()

z = df['P_INFANT'] * df['P_75+'].std(0) + 1
p = (df['P_INFANT'] * df['P_75+']).std(0)

print(m, z, p)

0.001064567335037682 0.001064567335037682 0.001064567335037682

%time
kids_kmeans = cuML.KMeans(n_clusters=10)
kids_kmeans.fit(df[fields])

CPU times: user 338 ms, sys: 74.9 ms, total: 413 ms
Wall time: 411 ms
```
RAPIDS Geospatial Applications

John Murray @MurrayData

“RAPIDS opens up new opportunities by simplifying the application of geographic data science at scale, at speed. Applications are limited only by your imagination.

“While we have achieved a lot with RAPIDS, in the short time since initial launch, I believe that we have only scratched the surface so far.”

John Murray, Geographic Data Science Lab, University of Liverpool
RAPIDS Geospatial Applications

I’ll walk 500 miles...

It’s [every walkable road in Great Britain] a sizeable graph consisting of 3,078,131 vertices and 7,347,806 edges so represents a significant mathematical challenge, so I used Graphics Processing Unit (GPU) computing.

https://www.citymetric.com/horizons/so-where-exactly-did-proclaimers-walk-500-miles-4629
Geospatial Challenges

Still much more to do

Data Representation & Management
Indexing, Database Queries, Aggregation & KPIs
Positioning & Navigation (Indoor, Outdoor)
Machine Learning, Big Data Analytics, Behavior Models Event Analytics & Anomaly Detection
Map-based Visualization
cuSpatial
1. Data Representation for point, line, polygon (Columinar/SoA)
2. Location Data Ingestion from JSON schema (IVA schema data)
3. Spatial window query
4. Point-in-polygon test
5. Converting lat/lon to x/y
6. Haversine Distance between pairs of lat/lon points
7. Location-to-trajectory
8. Computing trajectory distance/speed
9. Computing trajectory spatial bounding box
10. Directed Hausdorff distance
11. Python bindings for all the above features
12. Python test code, sample application & performance evaluation scripts
## cuSpatial

### Today and Tomorrow

<table>
<thead>
<tr>
<th>Layer</th>
<th>0.10/0.11 Functionality</th>
<th>Functionality Roadmap (2020)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level Analytics</td>
<td>C++ Library w. Python bindings enabling distance, speed, trajectory similarity, trajectory clustering</td>
<td>C++ Library w. Python bindings for additional spatio-temporal trajectory clustering, acceleration, dwell-time, salient locations, trajectory anomaly detection, origin destination, etc.</td>
</tr>
<tr>
<td>Graph layer</td>
<td>cuGraph</td>
<td>Map matching, Djikstra algorithm, Routing</td>
</tr>
<tr>
<td>Query layer</td>
<td>Nearest Neighbor, Range Search</td>
<td>KNN, Spatiotemporal range search and joins</td>
</tr>
<tr>
<td>Index layer</td>
<td>Grid, Quad Tree</td>
<td>R-Tree, Geohash, Voronoi Tessellation</td>
</tr>
<tr>
<td>Geo-operations</td>
<td>Point in polygon (PIP), Haversine distance, Hausdorff distance, lat-lon to xy transformation</td>
<td>Line intersecting polygon, Other distance functions, Polygon intersection, union</td>
</tr>
<tr>
<td>Geo-representation</td>
<td>Shape primitives, points, polylines, polygons</td>
<td>Additional shape primitives</td>
</tr>
</tbody>
</table>
# cuSpatial 0.10

## Performance at a Glance

<table>
<thead>
<tr>
<th>cuSpatial Operation</th>
<th>Input data</th>
<th>cuSpatial Runtime</th>
<th>Reference Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-in-Polygon Test</td>
<td>1.3+ million vehicle point locations and 27 Region of Interests</td>
<td>1.11 ms (C++) 1.50 ms (Python) [Nvidia Titan V]</td>
<td>334 ms (C++, optimized serial) 130468.2 ms (python Shapely API, serial) [Intel i7-7800X]</td>
<td>301X (C++) 86,978X (Python)</td>
</tr>
<tr>
<td>Haversine Distance Computation</td>
<td>13+ million Monthly NYC taxi trip pickup and drop-off locations</td>
<td>7.61 ms (Python) [Nvidia T4]</td>
<td>416.9 ms (Numba) [Nvidia T4]</td>
<td>54.7X (Python)</td>
</tr>
<tr>
<td>Hausdorff Distance Computation (for clustering)</td>
<td>10,700 trajectories with 1.3+ million points</td>
<td>13.5s [Quadro V100]</td>
<td>19227.5s (Python SciPy API, serial) [Intel i7-6700K]</td>
<td>1,400X (Python)</td>
</tr>
</tbody>
</table>
cuSpatial 0.10

Try It Today!

conda install -c rapidsai-nightly cuspatial
Community
Ecosystem Partners

CONTRIBUTORS

ADOPTERS

OPEN SOURCE
Building on top of RAPIDS
A bigger, better, stronger ecosystem for all

High-Performance Serverless event and data processing that utilizes RAPIDS for GPU Acceleration

GPU accelerated SQL engine built on top of RAPIDS

Distributed stream processing using RAPIDS and Dask
Explore: RAPIDS Code and Blogs

Check out our code and how we use it

https://github.com/rapidsai

https://medium.com/rapids-ai
Getting Started
RAPIDS
How do I get the software?

- https://github.com/rapidsai
- https://anaconda.org/rapidsai/
- https://hub.docker.com/r/rapidsai/rapidsai/
Join the Movement
Everyone can help!

APACHE ARROW
https://arrow.apache.org/  
@ApacheArrow

RAPIDS
https://rapids.ai  
@RAPIDSAI

Dask
https://dask.org  
@Dask_dev

GPU Open Analytics Initiative
http://gpuopenanalytics.com/  
@GPUOAI

Integrations, feedback, documentation support, pull requests, new issues, or code donations welcomed!
THANK YOU

Joshua Patterson  
@datametrician  
joshuap@nvidia.com